Quasi-delay-insensitive circuits are Turing-complete[†]

Rajit Manohar and Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125.

November 17, 1995

Abstract: *Quasi-delay-insensitive (QDI) circuits are those whose correct operation does not depend on the delays of operators or wires, except for certain wires that form isochronic forks. In this paper we show that quasi-delay-insensitivity, stability and non-interference, and strong confluence are equivalent properties of a computation. In particular, this shows that QDI computations are deterministic. We show that the class of Turing-computable functions have QDI implementations by constructing a QDI Turing machine.*

Keywords: Quasi-delay-insensitive circuits; Turing machines; Determinism; Confluence; Stability

1. Introduction

There has been a renewal of interest in the design of asynchronous circuits, motivated by the potential benefits of designing circuits in an asynchronous fashion. Asynchronous circuits exhibit average case behavior and can therefore be optimized in a data-dependent fashion. Another benefit is that the portion of the circuit involved in the computation is the only part that dissipates power. As a result, asynchronous design methods are relevant for applications where low power consumption is important.

Various models of CMOS circuits are used to hide the electrical properties of transistors, which would otherwise complicate the design process. These models typically assume that voltages represent boolean values, and that a transistor can be thought of as a switch. Delay-insensitive circuit design assumes that the correct operation of a circuit is independent of the delay in operators and wires. It was shown in [3] that the class of circuits that are entirely delay-insensitive is quite limited. Speed-independent circuit design assumes that operators can take an arbitrary amount of time to switch, but that wires have negligible delays compared to operators. Self-timed circuits assume that wires have negligible delay compared to gates in local isochronic regions [6]. Quasi-delay-insensitive circuit design assumes that both operators and wires can take an arbitrary time to switch, except for certain wires that form *isochronic forks* [2].

In this paper we show that the class of QDI circuits is Turing complete (modulo finite memory), i.e., any recursive function can be computed with QDI circuits. We show that state transitions in QDI circuits must exhibit the diamond property and, as a consequence, all QDI computations are entirely deterministic. In particular, this implies that one cannot build a QDI arbiter.

Strong confluence is closely related to the property of semi-modularity [5]. However, [5] only considers semi-modularity in the context of sequential machines. In addition, the computations considered are not semi-modular at every state, but only at some states. In [7], it is shown that hazard-free speed-independent asynchronous circuits are deterministic, but under a different gate model. In particular only AND, OR, NOT gates and C-elements are considered. In addition, gates that cannot be directly realized in CMOS are permitted in their model, since they allow gates with inverted inputs.

This paper is organized as follows. We introduce our circuit model and explain what a QDI circuit is in

[†] The research described in this report was sponsored by the Advanced Research Projects Agency and monitored by the Office of Army Research.

terms of this model. We then prove necessary and sufficient conditions for a circuit to be QDI, and discuss various consequences of this. We give the construction of a QDI Turing machine with a semi-infinite tape as a concrete demonstration of the Turing-completeness of this class of circuits.

2. Circuit model

A CMOS circuit is a network of *gates*, where each gate can have an arbitrary number of inputs and one output. We assume that all circuits are closed: each variable of the circuit is the input of a gate and the output of a gate. An open circuit can be transformed into a closed one by representing the environment of the circuit as gates.

The output of a gate is connected to the low voltage level (used to represent the boolean **false**) via a transistor network (known as the pull-down), and to the high voltage level (used to represent the boolean **true**) via another transistor network (known as the pull-up). These two transistor networks together form the gate, as shown in Fig. 1.

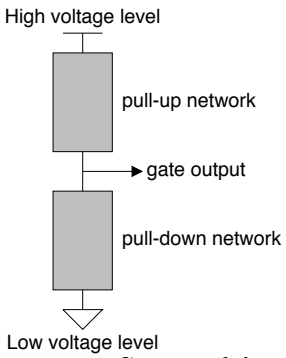

Fig. 1: Gate model.

A transistor is modelled as a switch that establishes or cuts electrical connections between nodes, depending on the voltage of the gate of the transistor. A pull-up/pull-down network is modelled as a network of switches that determine if two nodes are connected or disconnected. This network is represented as a boolean function which is true just when the two nodes of interest are connected.

Since the pull-up and pull-down networks are modelled as boolean functions, a gate can be represented by a pair of boolean functions. Such a representation can be expressed using *production rules* [2].

Definition. (*production rule*) (1)

A production rule (PR) is a construct of the form $G \mapsto t$, where t is a simple assignment (a transition), and G is a boolean expression known as the guard of the PR.

A gate with output x, pull-up network B^+, and pull-down network B^- corresponds to the production rules

$$B^+ \mapsto x\uparrow$$
$$B^- \mapsto x\downarrow$$

$x\uparrow$ and $x\downarrow$ are abbreviations for the assignments $x :=$ **true** and $x :=$ **false** respectively. We use the predicate R on transitions to denote the result of a transition: $R(x\uparrow) \equiv x$ and $R(x\downarrow) \equiv \neg x$. For example, a Muller C-element with inputs a and b and output c would be represented by production rules

$$a \wedge b \mapsto c\uparrow$$
$$\neg a \wedge \neg b \mapsto c\downarrow$$

Definition. (*production rule set*) (2)

A production rule set is the concurrent composition of all the production rules in the set.

A production rule set is used to describe a network of gates.

Definition. (*execution*) (3)

An execution of a production rule $G \mapsto t$ is an unbounded sequence of firings. A firing of $G \mapsto t$ when G is **true** *amounts to the execution of t, and a firing with G* **false** *amounts to a* **skip**. *The firing of a production rule in a state where $G \wedge \neg R(t)$ holds is said to be effective; otherwise, the firing is said to be vacuous. The execution of a production rule set corresponds to the weakly fair concurrent composition of the individual production rules in the set.*

Although one could assume that the transitions on wires are instantaneous, a CMOS circuit does not have this property. We make the weaker assumption that transitions on wires are monotonic. Since we make this assumption, we insist that no production rule is *self-invalidating*.

Definition. (*self-invalidating production rule*) (4)

A production rule $G \mapsto t$ is said to be self-invalidating when $R(t) \Rightarrow \neg G$.

A self-invalidating production rule corresponds to a gate whose output transition disabled itself.

If the output of a gate is at the low voltage level, it can change to the high voltage level when the pull-up network becomes conducting. This can happen as a result of the environment changing some input to the gate. If the input of the gate can change in a manner that makes the pull-up network non-conducting before the output of the gate changes, the circuit is said to exhibit a hazard since the circuit could have a glitch on the output of the gate.

Definition. (*non-interference*) (5)

The production rules $B^+ \mapsto x\uparrow$ and $B^- \mapsto x\downarrow$ are said to be non-interfering in a computation if and only if $\neg B^+ \vee \neg B^-$ is an invariant of the computation. A production rule set is non-interfering if every production rule in the set is non-interfering.

Let $B^+ \mapsto x\uparrow$ and $B^- \mapsto x\downarrow$ be a pair of production rules that define the gate for x. If $B^+ \wedge B^-$ were **true** at any point in the computation, the result at the circuit level would correspond to connecting the high voltage level to the low voltage level—a short circuit! Non-interference guarantees that such a short-circuit cannot occur. (Note that a CMOS circuit implementation will have short-circuit currents when a gate switches; however, these currents are transient.)

3. Quasi-delay-insensitive circuits

A circuit is said to be quasi-delay-insensitive if its correct operation is independent of the delays of gates and wires, except for certain wires that form isochronic forks.

3.1. Isochronic forks and inverters

A fork in a circuit corresponds to an output of a gate being used as the input for more than one gate. As an example, consider the fork in Fig. 2. The fork connects output x of gate G to the input $x1$ of gate $G1$ and the input $x2$ of gate $G2$.

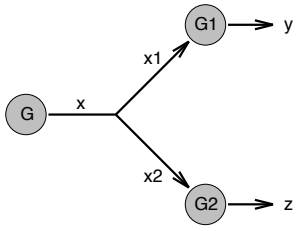

Fig. 2: An example of a fork.

The fork being isochronic means that some transitions on x are not acknowledged by a transition of both y and z—the outputs of gates $G1$ and $G2$ respectively.

For instance, transition $x\uparrow$ causes transitions $x1\uparrow$ and $x2\uparrow$. Transition $x1\uparrow$ causes (is acknowledged by) transition $y\uparrow$. But transition $x2\uparrow$ does not cause a transition on z. Hence, the completion of transition $x2\uparrow$ has to be justified by timing assumptions. We assume that, when transition $x1\uparrow$ has been acknowledged by transition $y\uparrow$, transition $x2\uparrow$ is also completed. This assumption is called the "isochronicity assumption." [3].

A transition on a variable, say x, can complete in two ways: the voltage of x reaches a value that causes the gate of which x is an input to switch (i.e. a transition on the output takes place); the voltage of x reaches a value that prevents the gate of which x is an input from switching.

In both cases, the voltage value for which the transition is considered completed depends on the structure of the gate—transistor and switching thresholds, in particular. We can abstract from specific physical dependencies by modelling the time behavior of the transition as a hypothetical "wire delay." It is this abstraction that allows us to say that forks are isochronic when the propagation delays on all branches of the fork are identical—hence the term "isochronic."

If all forks are isochronic, it is possible to lump the delays of all output branches of a gate into the delay of the gate and assume that all wires delays are zero. Therefore speed-independence is equivalent to assuming that all forks are isochronic, and thus fulfilling the isochronicity requirement is the most practical way of implementing speed-independence.

Fulfilling the isochronicity requirement is considered straightforward, except when there is an explicit inverter on the branch of the fork whose transition is not acknowledged. This means that there is an additional delay—namely, the time taken for an inverter to switch—added to the delay of the wire. Since we have already assumed that gates can take an arbitrary amount of time to switch, this implies that we can no longer meet the isochronicity requirement without making an additional timing assumption on gate delays—an assumption that we do not choose to make.

Therefore, the only gates we permit are those that do not need explicit inverters for their implementation. Such gates are said to be *directly implementable*. A production-rule representation of a gate $B^+ \mapsto x\uparrow$ and $B^- \mapsto x\downarrow$ corresponds to a directly implementable gate if and only if the negation-normal form of B^+ contains only inverted literals, and the negation normal form of B^- contains only noninverted literals. This is a consequence of using only P-transistors for pull-up networks and only N-transistors for pull-down networks.

The introduction of inverters to generate inverted versions of certain variables may result in a circuit that is no longer QDI! The process of determining where inverters should be placed and adjusting the senses of various signals to make a production-rule set directly implementable is known as *bubble reshuffling*. For instance, an AND gate with inputs a and b, and output c is described by the production rule set

$$\begin{aligned} a \wedge b &\mapsto c\uparrow \\ \neg a \vee \neg b &\mapsto c\downarrow \end{aligned}$$

To make this production rule set directly implementable, we can invert the sense of c. The negated version of c is written as $c\_$. We obtain

$$\begin{aligned} \neg a \vee \neg b &\mapsto c\_\uparrow \\ a \wedge b &\mapsto c\_\downarrow \end{aligned}$$

We can now add an inverter on the output to obtain c from $c\_$. This transformation does not affect the QDI property of the circuit because the rest of the circuit can never examine $c\_$. This observation is general: one can invert the sense of the output of a gate and add an inverter after it without affecting the QDI property of the circuit that uses the gate.

Suppose, instead, we decided to implement the AND gate by inverting the inputs to obtain

$$\neg a\_ \wedge \neg b\_ \mapsto c\uparrow$$
$$a\_ \vee b\_ \mapsto c\downarrow$$

If the uninverted senses of a and b are used in other parts of the circuit and the fork between this AND gate and the rest of the circuit is isochronic, then the introduction of inverters to generate $a\_$ and $b\_$ can result in a circuit that is no longer QDI.

There are tools that automatically determine where inverters can be placed so that the resulting production rule set is directly implementable [8].

3.2. Circuit malfunction

We assume that the only way a QDI circuit can malfunction is if the output of any gate in the circuit glitches. If all gates are hazard-free, then we consider the circuit to be QDI. (An error in the design of a circuit may produce a QDI circuit that implements a different specification!)

Definition. (*stability*) (6)

A production rule $G \mapsto t$ is said to be stable in a computation if and only if G can change from **true** *to* **false** *only in those states of the computation in which $R(t)$ holds. A production rule set is said to be stable if and only if every production rule in the set is stable.*

Note that stability is not guaranteed by the implementation of a single gate, but is a property of the entire computation. Martin's synthesis method compiles CSP programs into production rules. The synthesis procedure guarantees that the resulting production rule set is stable and non-interfering.

Theorem. (*quasi-delay-insensitivity*) (7)

A circuit is QDI if and only if the production rule set describing it is stable and non-interfering.

Proof: Suppose the production rule set is unstable. Then, there exists a gate represented by the production rules $B^+ \mapsto z\uparrow$ and $B^- \mapsto z\downarrow$ with an unstable production rule. Without loss of generality, there is a state in which $\neg z \wedge B^+$ holds, which is followed by a state in which $\neg z \wedge \neg B^+$ holds before z changes. Therefore, the output of the gate can glitch, which implies that the circuit is not QDI. If the production rule set is non-interfering, there can be a short-circuit.

Suppose the production rule set is stable and non-interfering. Consider a gate $B^+ \mapsto z\uparrow$ and $B^- \mapsto z\downarrow$. From stability, we know that if $\neg z \wedge B^+$ holds, then B^+ remains true until z changes. In other words, we cannot have a state in which $\neg z \wedge \neg B^+$ holds before z changes. Similarly, the transition $z\downarrow$ is also hazard-free, implying that the gate is hazard-free. Since every gate is hazard-free, the circuit is QDI. □

4. Confluence, Determinism, and Arbiters

In this section we examine some of the consequences of stability and non-interference, the two properties that characterize QDI computations. The following definition can be found in [1].

Definition. (*strong confluence*) (8)

Let t_1 and t_2 be two transitions that can fire in state s. Let s_1 be the state obtained by firing t_1 in s, and s_2 be the state obtained by firing t_2 in s. The computation is said to be strongly confluent, if t_1 can fire in state s_2 and t_2 can fire in state s_1, and both alternatives lead to the same final state. (cf. Fig. 3)

Theorem. (*strong confluence*) (9)

A computation can be described by a stable, non-interfering production rule set if and only if it is strongly confluent.

Proof: Let $G_1 \mapsto t_1$ and $G_2 \mapsto t_2$ be two production rules that have effective firings in state s, i.e., $s \Rightarrow G_1 \wedge G_2 \wedge \neg R(t_1) \wedge \neg R(t_2)$. Now, t_1 cannot make G_2 false, since that would make the production rule unstable. Therefore, after t_1 fires, $G_2 \mapsto t_2$ can still fire. Similarly, $G_1 \mapsto t_1$ can fire after t_2 changes as well. Since all transitions are elementary assignments, the final state does not depend on the order of the two firings. Therefore, the computation is strongly confluent.

Conversely, suppose a computation is strongly confluent. For each transition t, define $G(t)$ as the disjunction of all the states in which the transition has an effective firing. Then we claim that the production rule set $\{G(t) \mapsto t \mid t \textit{ is a transition}\}$ is a stable, non-interfering production rule set that describes the computation. Let $G(t) \mapsto t$ be a production rule that has an effective firing in some state s. Firing any other production rule cannot disable t, since that would violate strong confluence. This implies that the rule $G(t) \mapsto t$ is stable. Since $G(t) \mapsto t$ is an arbitrary production rule, the production rule set is stable. The production rule set is non-interfering since both $x\uparrow$ and $x\downarrow$ cannot have an effective firing in a state s, which implies that $G(x\uparrow) \wedge G(x\downarrow) \equiv \textbf{false}$. Finally, the production rule set correctly describes the computation since, by construction, a transition is enabled in the production rule set if and only if the transition had an effective firing in the original computation. □

Theorem (9) does not directly rule out self-invalidating production rules. However, these rules can be systematically eliminated by the introduction of new variables. Let one of $B^+ \mapsto x\uparrow$ and $B^- \mapsto x\downarrow$ be a self-invalidating rule. We can replace these rules with the following (y is fresh):

$$\begin{array}{llll} B^+ & \mapsto & y\uparrow & \qquad y \mapsto x\uparrow \\ B^- & \mapsto & y\downarrow & \qquad \neg y \mapsto x\downarrow \end{array}$$

These rules are no longer self-invalidating since y is a fresh variable. They also do not change the result of the computation. Therefore, ruling out self-invalidating rules does not restrict the computation in any way. (The rules $y \mapsto x\uparrow$ and $\neg y \mapsto x\downarrow$ are implemented with two inverters.)

Consider any strongly confluent computation. Suppose we identify all the effective firings that can take place at a particular point in the execution and artificially prevent any other production rule from firing. Then, no matter which path was taken by the computation, the final result would be the same. This observation holds at any point in the computation. We conclude that a strongly confluent computation is essentially deterministic. Therefore, a QDI computation will always be deterministic.

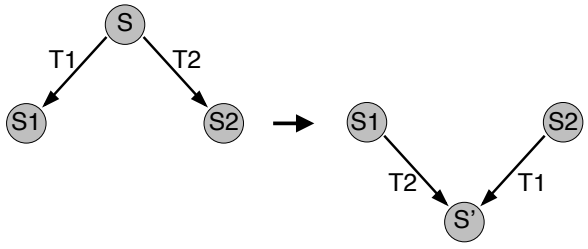

Fig. 3: Strongly confluent computation.

An arbiter with inputs ai and bi, and outputs u and v is described by the handshaking expansion

$$\begin{array}{l} *[[ai \longrightarrow u\uparrow; [\neg ai]; u\downarrow \\ \quad | \, bi \longrightarrow v\uparrow; [\neg bi]; v\downarrow \\ ]] \end{array}$$

where the thin bar for the selection denotes arbitration. There is no QDI implementation of this circuit because a computation that uses an arbiter cannot be strongly confluent. In the state in which $ai \wedge bi$ holds, both $u\uparrow$ and $v\uparrow$ can fire. However, after $u\uparrow$ fires, $v\uparrow$ can no longer fire. This implies that when designing an arbiter, we have to consider the electrical behavior of transistor circuits.

5. Compilation of a Turing machine

In this section we demonstrate that QDI circuits can be used to compute any computable function by constructing a QDI bounded-tape Turing machine. Since any computation can only use a finite amount of memory (since any physical implementation of a computation can only use finite resources), this demonstrates that restricting the design space to QDI circuits does not limit the class of functions that can be computed. The implementation we propose does not include any inverters on a single branch of a fork, and therefore doesn't contain inverters on the branch of an isochronic fork. Hence, the implementation is QDI (and therefore, also speed-independent).

Let $TM = \langle S, K, \delta \rangle$ be a Turing machine with a semi-infinite tape where S and K are positive integers, and δ is a function

$$\delta\colon \{q_0, \ldots, q_S\} \times \{\sigma_0, \ldots, \sigma_K\} \rightarrow \{q_0, \ldots, q_S\} \times \{\sigma_0, \ldots, \sigma_K, L, R\}$$

satisfying $\delta(q_1, \sigma_k) = (q_1, \sigma_k)$ for all k. Using s to denote the state, array a to denote the tape, and p to denote the head position, a CSP program that describes such a Turing machine is:

$$\begin{array}{l} TM \equiv \;\; s := q_0; p := 0; \\ \quad *[\;\; (s,d) := \delta(s, a[p]); \\ \quad\quad [\;\; d = L \longrightarrow p := p - 1 \;\; [] \;\; d = R \longrightarrow p := p + 1 \;\; [] \;\; else \longrightarrow a[p] := d \;\;] \\ \quad] \end{array}$$

The Turing machine is initialized in state zero with its head located at position zero on the tape. It uses δ, commonly referred to as the *next move* function to determine the action it should take. It then updates the tape appropriately, and continues the computation. In the rest of this section, we omit the assignments $s := q_0$ and $p := 0$, since they can be performed by appropriately initializing the circuit on reset.

5.1. Process Decomposition

The computation of the Turing machine proceeds in two steps. Using the current value on the tape and the current value of the state, the next state and action is computed. Following this, the tape is appropriately updated. We therefore decompose the Turing machine into two parts, one for each distinct step of the computation. Using variable c to denote the current value of the symbol on the tape, we obtain:

$$\begin{array}{l} TM1 \equiv *[\;\; Tout?c; \;\; (s,d) := \delta(s,c); \;\; Tin!d \;\;] \\ TM2 \equiv *[\;\; Tout!a[p]; \;\; Tin?d; \\ \quad\quad [\;\; d = L \longrightarrow p := p - 1 \;\; [] \;\; d = R \longrightarrow p := p + 1 \;\; [] \;\; else \longrightarrow a[p] := d \;\;] \\ \quad] \\ TM \equiv \;\; TM1 \;\; \| \;\; TM2 \end{array}$$

The computation $(s, d) := \delta(s, c)$ reads and writes the same variable s. Since we cannot perform this operation without making a temporary copy of s, we make this copy explicit by introducing a state buffer. Using $(X?)$ as an abbreviation for the value received on X, we obtain:

$$\begin{array}{l} next \equiv *[\;\; (s,d) := \delta(Sout?, Tout?); \;\; Sin!s, Tin!d \;\;] \\ statebuf \equiv *[\;\; Sout!x; Sin?x \;\;] \\ TM1 \equiv \;\; next \;\; \| \;\; statebuf \end{array}$$

We do not decompose $TM1$ furthur, since both *next* and *statebuf* can easily be transformed into a circuit (see below). For the remainder of this section, we concentrate on $TM2$.

A computation resembles a function block when it is of the form "read inputs", followed by "produce outputs." This computation has a standard QDI implementation [4]. We make $TM2$ resemble this form by introducing a tape buffer process.

$$
\begin{aligned}
&\mathit{fulltape} \equiv *[\ \mathit{Tin}?d;\\
&\qquad [\ d = L \longrightarrow p := p-1\ \ []\ \ d = R \longrightarrow p := p+1\ \ []\ \ \mathit{else} \longrightarrow a[p] := d\ \ ];\ \ \mathit{TTout}!a[p]\\
&\quad]\\
&\mathit{tapebuf} \equiv *[\ \mathit{Tout}!x; \mathit{TTout}?x\ ]\\
&TM2 \equiv\ \mathit{fulltape}\ \|\ \mathit{tapebuf}
\end{aligned}
$$

On receiving an input on Tin, *fulltape* updates its current state by either changing the value of p or changing the value of $a[p]$. Finally, the value of $a[p]$ is read. Since reading $a[p]$ and modifying p and $a[p]$ happen in sequence, we can implement this sequencing once and write *fulltape* as a reactive process. We split the tape into the tape control, which sequences the read and update operations, and the tape itself.

$$
\begin{aligned}
&\mathit{tapecontrol} \equiv *[\ \mathit{LTin}!(\mathit{Tin}?); \mathit{TTout}!(L?)\ ]\\
&\mathit{tape} \equiv *[[\overline{\mathit{LTin}} \longrightarrow \mathit{LTin}?d;\ \ [d = L \longrightarrow p := p-1\ \ []\ \ d = R \longrightarrow p := p+1\ \ []\ \ \mathit{else} \longrightarrow a[p] := d]\\
&\qquad []\overline{L} \longrightarrow L!a[p]\\
&\quad]]\\
&\mathit{fulltape} \equiv\ \mathit{tapecontrol}\ \|\ \mathit{tape}
\end{aligned}
$$

To implement *tape*, we consider the tape to be the concurrent composition of a linear array of tape elements as shown in Fig. 4. Each tape element maintains information about its position relative to the tape head. Its current state s is l if the element is to the left of the head position, r if the element is to the right of the head position, and t otherwise. At any point, the tape state (from left to right) can be described by the regular expression l^+tr^*, where the length of the expression is the size of the tape. (The presence of a leading "l" is guaranteed by the program of a Turing machine with a semi-infinite tape.)

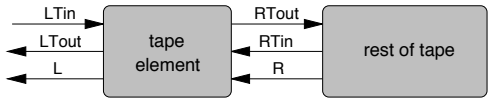

Fig. 4: Decomposition of tape into an array of tape elements.

Each tape element contains a register that stores the symbol in the tape element. Apart from this register, each tape element must maintain its state, s. The operation of a tape element (given its current state) can be described as follows:

- $s = \mathtt{l}$. If a "move left" action is to be performed, it is communicated to the rest of the tape. The new state is the state of the first process in the rest of the tape. If a write, read, or a "move right" action is performed, this action is communicated to the rest of the tape, and the state remains unchanged.
- $s = \mathtt{r}$. A "move left", read, and write action can never happen. If a "move right" action occurs, the new state is `t`.
- $s = \mathtt{t}$. A "move left" action results in state `r`. A "move right" action results in state `l`, and this action is communicated to the rest of the tape so as to move the head to the right. A read/write action is sent to the register.

Since the next state of a tape element can depend on the state of the first element in the rest of the tape, we introduce channels *RTin* and *LTout* that communicate this information.

Once again, since the tape reads and modifies its own state, we introduce a buffer to make the copy explicit. The CSP description of the tape element is:

$$
\begin{aligned}
tapeelem \equiv *[[&[\overline{LTin} \longrightarrow LTin?d, STin?s;\\
&\quad [d = L \longrightarrow [s = \mathtt{l} \longrightarrow RTout!d; RTin?s; LTout!\mathtt{l}\\
&\qquad\quad [] s = \mathtt{t} \longrightarrow LTout!\mathtt{t}; s := \mathtt{r}\\
&\qquad\quad]\\
&\quad [] d = R \longrightarrow [s = \mathtt{l} \longrightarrow RTout!d; RTin?; LTout!s\\
&\qquad\quad [] s = \mathtt{t} \longrightarrow RTout!d; RTin?; LTout!s; s := \mathtt{l}\\
&\qquad\quad [] s = \mathtt{r} \longrightarrow LTout!s; s := \mathtt{t}\\
&\qquad\quad]\\
&\quad [] else \longrightarrow [s = \mathtt{l} \longrightarrow RTout!d; RTin?; LTout!s\\
&\qquad\quad [] s = \mathtt{t} \longrightarrow GW!d; LTout!s\\
&\qquad\quad]\\
&\quad];\ STout!s\\
&\ [] \overline{L} \longrightarrow [s = \mathtt{l} \longrightarrow L!(R?)\\
&\qquad [] s = \mathtt{t} \longrightarrow L!(GR?)\\
&\qquad]\\
&]]
\end{aligned}
$$

$$
\begin{aligned}
tapereg \equiv *[[&\overline{GR} \longrightarrow GR!x\\
&[] \overline{GW} \longrightarrow GW?x\\
&]]
\end{aligned}
$$

$$tapestate \equiv *[STin!x; STout?x]$$

$$tapeelement \equiv \ tapeelem \ \| \ tapereg \ \| \ tapestate$$

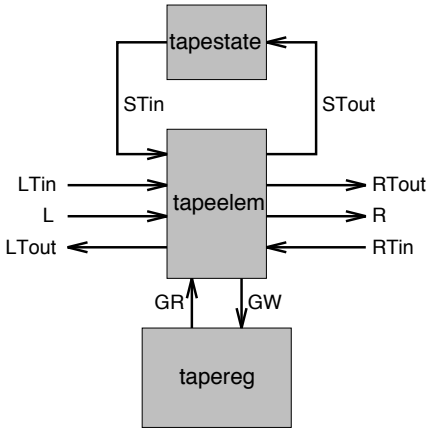

Fig. 5: Decomposition of a tape element.

Since we need an additional channel *LTout* to perform correct state updates, we modify *tapecontrol* to the following process:

$$tapecontrol \equiv *[\ LTin!(Tin?); TTout!(L?), LTout?\ ]$$

The complete tape element decomposition is shown in Fig. 5.

In the following sections, we compile the Turing machine into production rules. The compilation will proceed by translating each CSP process into handshaking expansions and, finally, into directly implementable production rules.

5.2. Compilation of *TM1*

The handshaking expansion for $TM1$ is straightforward. We use the output on Sin and Tin as the acknowledge for $Sout$ and $Tout$, and the input on $Sout$ and $Tout$ as the request for data on Sin and Tin. We have:

$next \equiv *[[v(Sout) \wedge v(Tout)]; Sin\Uparrow, Tin\Uparrow; [n(Sout) \wedge n(Tout)]; Sin\Downarrow, Tin\Downarrow]$

$statebuf \equiv *[Sout := a; [v(Sin)]; a := Sin; Sout\Downarrow; [n(Sin)]]$

where the functions v and n encode the validity and neutrality test respectively. We provide the production rules corresponding to a one-bit version of *statebuf*. This construction can be easily generalized to n-bits.

$$statebuf1bit \equiv *[\ [\neg Sint \wedge \neg Sinf];\ \ [at \longrightarrow Soutt\uparrow\ []\ af \longrightarrow Soutf\uparrow];$$
$$[Sint \longrightarrow af\downarrow; at\uparrow\ []\ Sinf \longrightarrow at\downarrow; af\uparrow];\ \ Soutt\downarrow, Soutf\downarrow\ ]$$

The production rules are:

$$\neg Sint \wedge \neg Sinf \wedge \neg af \ \mapsto\ Soutt\uparrow \qquad at \vee Sint \ \mapsto\ af\downarrow$$
$$\neg Sint \wedge \neg Sinf \wedge \neg at \ \mapsto\ Soutf\uparrow \qquad \neg Sinf \wedge \neg af \ \mapsto\ at\uparrow$$

$$(Sint \wedge at) \vee (Sinf \wedge af) \ \mapsto\ Soutt\downarrow \qquad af \vee Sinf \ \mapsto\ at\downarrow$$
$$(Sint \wedge at) \vee (Sinf \wedge af) \ \mapsto\ Soutf\downarrow \qquad \neg Sint \wedge \neg at \ \mapsto\ af\uparrow$$

Notice that the compilation of this buffer completes the compilation of *tapebuf*, and part of *tapeelement* as well. The rest of $TM1$ can be compiled using the standard *function block* compilation technique [4], and depends on the next move function δ. The block diagram of $TM1$ is shown in Fig. 6.

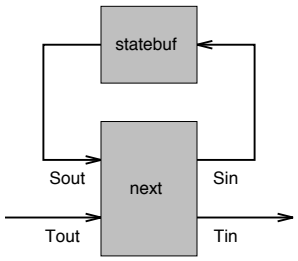

Fig. 6: Compilation of *TM1*.

5.3. Compilation of the tape

To simplify the handshaking expansion for the tape, we use the input on $LTin$ as a request for data on $LTout$, and the output on $LTout$ as the acknowledge for channel $LTin$. The protocol used on L is the usual four-phase handshake.

Compiling tapecontrol. The handshaking expansion for *tapecontrol* is given by:

$$*[[v(Tin)]; LTin := Tin; [v(LTout)]; LTin\Downarrow; [n(LTout)];$$
$$Lo\uparrow; [v(Li)]; TTout := Li; [n(Tin)]; Lo\downarrow; [n(Li)]; TTout\Downarrow]$$

We use *process factorization* to split the handshaking expansion into the following two concurrent processes.

$$*[[v(Tin)]; LTin := Tin; [v(LTout)]; LTin\Downarrow; [n(LTout)]; Lo\uparrow; [n(Tin)]; Lo\downarrow]$$
$$\|$$
$$*[[v(Li)]; TTout := Li; [n(Li)]; TTout\Downarrow]$$

The second process can be translated into a number of wires. We compile the first process for one-bit data. This construction can be easily generalized for n-bits of data. We introduce a state variables st and sf to remove indistinguishable states. (We could have just used one variable, but the resulting production rule

set would need a number of extra inverters to make it directly implementable.) The resulting handshaking expansion is:

$$*[[\,Tint \longrightarrow LTint\uparrow \;[]\; Tinf \longrightarrow LTinf\uparrow\,];\;\; [\,LToutt \vee LToutf\,]; sf\downarrow; st\uparrow; LTint\downarrow, LTinf\downarrow;$$
$$[\,\neg LToutt \wedge \neg LToutf\,]; Lo\uparrow; st\downarrow; sf\uparrow; [\,\neg Tint \wedge \neg Tinf\,]; Lo\downarrow]$$

The production rules corresponding to this handshaking expansion are:

$$\neg Tint\_ \wedge \neg st \wedge \neg Lo \mapsto LTint\uparrow \qquad\qquad LToutt \vee LToutf \mapsto sf\downarrow$$
$$\neg Tinf\_ \wedge \neg st \wedge \neg Lo \mapsto LTinf\uparrow \qquad\qquad \neg LToutt \wedge \neg LToutf \wedge \neg st \mapsto sf\uparrow$$

$$st \mapsto LTint\downarrow \qquad\qquad \neg LToutt \wedge \neg LToutf \wedge \neg sf \mapsto Lo\uparrow$$
$$st \mapsto LTinf\downarrow \qquad\qquad Tint\_ \wedge Tinf\_ \wedge sf \mapsto Lo\downarrow$$

$$\neg sf \wedge \neg Lo \mapsto st\uparrow$$
$$Lo \mapsto st\downarrow$$

Notice that we have used the inverted sense of Tin ($Tin\_$) in this production rule set. We have to use an inverter to generate this signal from Tin.

Compiling a register. Compilation of the register used in *tapelement* is very simple. We compile a one-bit dual-railed register; the construction can be easily extended to n-bits. The handshaking expansion for the register is:

$$*[[\,GRi \wedge xt \longrightarrow GRto\uparrow; [\,\neg GRi\,]; GRto\downarrow$$
$$[]\,GRi \wedge xf \longrightarrow GRfo\uparrow; [\,\neg GRi\,]; GRfo\downarrow$$
$$[]\,GWti \longrightarrow xf\downarrow; xt\uparrow; GWo\uparrow; [\,\neg GWti\,]; GWo\downarrow$$
$$[]\,GWfi \longrightarrow xt\downarrow; xf\uparrow; GWo\uparrow; [\,\neg GWfi\,]; GWo\downarrow$$
$$]]$$

Since the register is accessed in a manner which guarantees mutual exclusion between GR and GW, the production rules for this process are given by:

$$\neg xf \wedge \neg GWfi \mapsto xt\uparrow \qquad\qquad \neg xt \wedge \neg GRi\_ \mapsto GRfo\uparrow$$
$$xf \vee GWfi \mapsto xt\downarrow \qquad\qquad GRi\_ \mapsto GRfo\downarrow$$

$$\neg xt \wedge \neg GWti \mapsto xf\uparrow \qquad\qquad (xf \wedge GWfi) \vee (xt \wedge GWti) \mapsto GWo\_\downarrow$$
$$xt \vee GWti \mapsto xf\downarrow \qquad\qquad \neg GWfi \wedge \neg GWti \mapsto GWo\_\uparrow$$

$$\neg xf \wedge \neg GRi\_ \mapsto GRto\uparrow$$
$$GRi\_ \mapsto GRto\downarrow$$

The production rule set uses the inverted sense of GRi ($GRi\_$) and generates the inverted sense of GWo ($GWo\_$). Since $GWo\_$ is an output, we can safely invert it to generate GWo.

Compiling the tape element control. The tape element control can send d to the right or to the register, and conditionally compute the new state. The handshaking expansion for the tape element control can be written as follows:

$$tapeelem1 \equiv *[[\,v(LTin) \wedge v(STin)\,];\;\; [\,R \longrightarrow RTout\Uparrow[]\neg R \longrightarrow skip\,], [\,W \longrightarrow GWo\Uparrow[]\neg W \longrightarrow skip\,], I\Uparrow$$
$$[((R \wedge v(RTout)) \vee (W \wedge GWi) \vee (\neg R \wedge \neg W)) \wedge v(I)\,]; STout\Uparrow, LTout\Uparrow;$$
$$[\,n(LTin) \wedge n(STin)\,]; RTout\Downarrow, GWo\Downarrow, I\Downarrow$$
$$[\,n(RTout) \wedge \neg GWi \wedge n(I)\,]; STout\Downarrow, LTout\Downarrow]$$

$$\begin{aligned}read \equiv *[[&Li \wedge s = \mathtt{l} \longrightarrow Ro\uparrow; [v(Ri)]; Lo\Uparrow; [\neg Li]; Ro\downarrow; [n(Ri)]; Lo\Downarrow\\ [\!]&Li \wedge s = \mathtt{t} \longrightarrow GRo\uparrow; [v(GRi)]; Lo\Uparrow; [\neg Li]; GRo\downarrow; [n(GRi)]; Lo\Downarrow\\ ]]&\end{aligned}$$

We can decompose the tape element control into the read and write part since the environment guarantees that they do not overlap. We use process factorization on *tapeelem*1 to decompose it into the following two processes, using channel I to communicate information needed by the other process.

$$\begin{aligned}change \equiv *[[&v(LTin) \wedge v(STin)]; [R \longrightarrow RTout\Uparrow [\!] \neg R \longrightarrow skip], [W \longrightarrow GWo\Uparrow [\!] \neg W \longrightarrow skip], I\Uparrow;\\ &[n(LTin) \wedge n(STin)]; RTout\Downarrow, GWo\Downarrow, I\Downarrow\ ]\end{aligned}$$

$$\begin{aligned}right \equiv *[[&v(I)]; [((R \wedge v(RTout)) \vee (W \wedge GWi)) \vee (\neg W \wedge \neg R)]; LTout\Uparrow, STout\Uparrow;\\ &[n(I) \wedge n(RTout) \wedge \neg GWi]; LTout\Downarrow, STout\Downarrow]\end{aligned}$$

We use two dual-railed bits (st_0, sf_0) and (st_1, sf_1) to encode s. The different states are:

$(s = \mathtt{l}) \equiv sf_0 \wedge sf_1$

$(s = \mathtt{t}) \equiv sf_0 \wedge st_1$

$(s = \mathtt{r}) \equiv st_0 \wedge sf_1$

The value received on $LTin$ is encoded using $2+2\lceil \log_2 K+1 \rceil$ wires. The encoding has the following meaning:

$LTin_0$ is true just when $d = L$.

$LTin_1$ is true just when $d = R$.

$LTin_2..LTin_{1+2\lceil \log_2 K+1 \rceil}$ are used to specify the dual-rail encoding of the symbol to be printed.

The information sent on channel I is used to determine the new state and acknowledge. This channel is encoded using 13 wires as follows:

$I_0..I_3$ contain the old state, which is used to generate the acknowledge.

$I_4..I_7$ contain the new state, if computable without information from $RTin$.

I_8 is true if the new state should be computed from the information received from $RTin$.

$I_9..I_{10}$ contains the dual-rail encoding of whether or not a communication was initiated on $RTout$.

$I_{11}..I_{12}$ contains the dual-rail encoding of whether or not a communication was initiated on GW.

(This encoding is not meant to be efficient in any way!) The predicates R and W are given by:

$R \equiv (s = \mathtt{l}) \vee (s = \mathtt{t} \wedge LTin = R)$

$W \equiv (LTin \neq L \wedge LTin \neq R) \wedge (s = \mathtt{t})$

Since both *tapeelem*1 and *right* are function blocks, they can be compiled using the standard technique outlined in [4]. We compile *read* assuming a one-bit dual-railed input on L; this compilation can be easily generalized to n-bits, for any n. The handshaking expansion for *read* is:

$$\begin{aligned}*[[&Li \wedge sf_0 \wedge sf_1 \longrightarrow Ro\uparrow; [Rti \longrightarrow Lto\uparrow [\!] Rfi \longrightarrow Lfo\uparrow]; [\neg Li]; Ro\downarrow;\\ &\quad [\neg Rti \wedge \neg Rfi]; Lto\downarrow, Lfo\downarrow\\ [\!]&Li \wedge sf_0 \wedge st_1 \longrightarrow GRo\uparrow; [GRti \longrightarrow Lto\uparrow [\!] GRfi \longrightarrow Lfo\uparrow]; [\neg Li]; GRo\downarrow;\\ &\quad [\neg GRti \wedge \neg GRfi]; Lto\downarrow, Lfo\downarrow\\ ]]&\end{aligned}$$

The production rules are:

$$\begin{aligned} Li \wedge sf_0 \wedge sf_1 &\mapsto Ro\_\downarrow & \neg Li &\mapsto Ro\_\uparrow\\ Li \wedge sf_0 \wedge st_1 &\mapsto GRo\_\downarrow & \neg Li &\mapsto GRo\_\uparrow\\ \\ Rti \vee GRti &\mapsto Lto\_\downarrow & \neg Rti \wedge \neg Rfi \wedge \neg GRti \wedge \neg GRfi &\mapsto Lto\_\uparrow\\ Rfi \vee GRfi &\mapsto Lfo\_\downarrow & \neg Rti \wedge \neg Rfi \wedge \neg GRti \wedge \neg GRfi &\mapsto Lfo\_\uparrow \end{aligned}$$

Note that we have generated the inverted signals *Lto_*, *Lfo_*, *Ro_* and *GRo_*. Since these are output variables, and are not used by any operators in this process, we can safely invert them to generate *Lto*, *Lfo*, *Ro*, and *GRo*. The entire *tapeelement* with the state buffer is shown in Fig. 7.

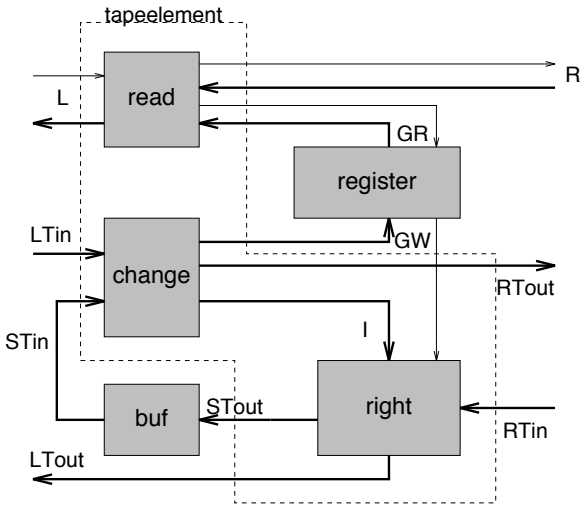

Fig. 7: Compilation of *tapeelement* as a number of function blocks.

5.4. Putting the pieces together

Each component given above is QDI. Some components require the inverted senses of certain signals as input. As noted in the previous section, the introduction of inverters can compromise the QDI of a circuit.

Observe that the inverted senses of signals are required by *tapecontrol* and the register. However, the inputs to these processes are not forked to any other process or operator! As a result, we can safely insert inverters between the processes to obtain a QDI Turing machine (cf. Fig. 8).

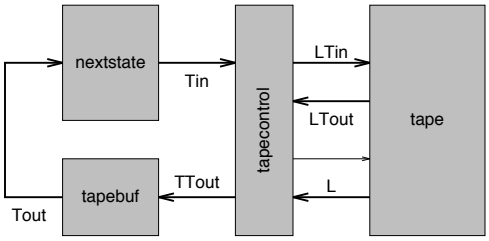

Fig. 8: A complete Turing machine.

By the construction given above, we can state that

Theorem. (*Turing-completeness*) (10)

Any bounded-tape Turing-computable function can be implemented using a QDI circuit.

6. Conclusions

We have shown that quasi-delay-insensitivity, stability and non-interference, and strong confluence are equivalent properties of a computation. We constructively demonstrated that, despite these restrictions on QDI computations, any Turing-computable function has a QDI implementation. This construction used only those gates that have a direct CMOS implementation.

References

[1] Leeuwen, J. van. *Handbook of Theoretical Computer Science, Volume B: Formal models and semantics.* MIT Press, 1990.

[2] Martin, Alain J. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing* **1**(4):226–234, 1986.

[3] Martin, Alain J. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conference on Advanced Research in VLSI*, 1990.

[4] Martin, Alain J. Asynchronous Datapaths and the Design of an Asynchronous Adder. *Formal Methods in System Design* **1**(1):119–137, July 1992.

[5] Miller, Raymond E. *Sequential Circuits and Machines.* John Wiley and Sons, 1965.

[6] Seitz, Charles L. Self-Timed VLSI Systems. *Proceedings of the 1st Caltech Conference on Very Large Scale Integration*, 1979.

[7] Smith, S.F. and Zwarico, A.E. Correct Compilation of Specifications to Deterministic Asynchronous Circuits. *Formal Methods in System Design* **7**:155-226, 1995.

[8] Lee, Tak-Kwan. Man page for `bubble`. Caltech Asynchronous Synthesis Tools, 1993.